Complex Intell. Syst. (2015) 1:69–80
DOI 10.1007/s40747-015-0003-3

ORIGINAL ARTICLE

Scalable distributed implementation of a biologically inspired parallel model

Gabriel Ciobanu[1]

Received: 19 August 2014 / Accepted: 6 November 2015 / Published online: 9 February 2016

Abstract The paper presents first the formal semantics of a parallel formalism inspired by biological cells, and then provides a faithful parallel implementation of this computational model using a known distributed computing middleware and taking care of various synchronization issues. Synchronization is achieved using barriers and preconditions; both refer to the fact that a membrane can apply its rules only after it has received signals from the other related membranes. A scalable parallel implementation is developed by using the MapReduce paradigm in GridGain which allows the splitting of a task into multiple subtasks, the parallel execution of these subtasks in parallel and the aggregation of the partial results into a single, final result. This implementation is appropriate for the description of this bio-inspired parallel model, a model which is computationally equivalent to Turing machines and able to provide polynomial solutions to NP-complete problems.

Keywords Bio-inspired parallel model · Membrane computing · Semantics · Scalable implementation

Introduction

Membrane systems are essentially parallel and nondeterministic computing models inspired by the compartments of (eukaryotic) cells and their biochemical reactions. The structure of a cell is represented by a set of hierarchically embedded membranes, all of which are contained inside a skin membrane. The molecular species (ions, proteins, etc.) floating inside and between cellular compartments are represented by multisets of objects described by means of symbols over a given alphabet. Chemical reactions are represented by evolution rules which operate on the objects, as well as on the compartmentalized structure (by dissolving, dividing, creating, or moving membranes). Membrane systems (also called P systems) perform parallel computations in the following way: starting from an initial configuration (the initial membrane structure and the initial multisets of objects placed inside the membranes), a system evolves by applying the evolution rules of each membrane in a nondeterministic manner. A rule is applicable when all the objects which appear in its left-hand side are available in the membrane where the rule is placed.

Since membrane systems aim to abstract the functioning of living cells, several extensions come from both cell biology and computer science. The computability power and efficiency have been investigated using the approaches of formal languages, automata and complexity theory. Membrane systems are presented together with many variants and examples in [10]. Several applications of these systems are presented in [7]. The state of the art is presented in the handbook published recently by Oxford University Press [11].

In this paper, we present a parallel implementation of membrane systems using GridGain [12], a JVM-based application middleware that supports the building of highly scalable real-time and data intensive distributed applications working on any infrastructure, from a small local cluster to large private grids and huge private, public and hybrid clouds. The implementation using such an appealing distributed computing technology involves some specific synchronization issues studied after defining the operational semantics and describing the parallel (sub)steps of evolution.

✉ Gabriel Ciobanu
gabriel@info.uaic.ro

[1] Romanian Academy, Institute of Computer Science, Iaşi, Romania

Operational semantics for membranes

In the basic model of membrane computing, objects are represented using symbols from a given alphabet, and each symbol from this alphabet can appear inside a region in many different copies. A membrane system is composed of membranes which do not intersect, and which are all contained within a skin membrane. Each membrane can contain multisets of objects, evolution rules and other membranes. The objects inside a membrane evolve in a maximal parallel manner according to the evolution rules inside the same membrane. According to [10], maximal parallel "means that we assign objects to rules, nondeterministically choosing the objects and the rules, until no further assignment is possible." Essentially, a *membrane system* of degree m is $\Pi = (O, \mu, w_1, \ldots w_m, (Rules(1), \rho_1), \ldots, (Rules(m), \rho_m), i_o)$, where:

- O is an alphabet of objects;
- μ is a membrane structure, with the membranes labeled by natural numbers $1 \ldots m$ in a one-to-one manner;
- w_i are the initial multisets over O associated with the membranes $1 \ldots m$;
- $Rules(1), \ldots, Rules(m)$ are finite sets of rules associated with the membranes $1 \ldots m$; the rules have the form $u \to v$, where u is a non-empty multiset of objects and v is a multiset containing messages which are of the form $(a, here)$, (a, out), (a, in_j) and the dissolving symbol δ;
- ρ_i is a partial order relation over R_i, specifying a priority relation among the rules: $(r_1, r_2) \in \rho_i$ iff $r_1 > r_2$ (i.e., r_1 has a higher priority than r_2);
- i_0 is either a number between 1 and m specifying the output membrane of Π, or it is equal to 0 indicating that the output is the outer region.

For a rule of form $u \to v$, the message $(a, here)$ in v says that a, once created, remains in the current membrane; (a, out) says that a, once created, is sent into the parent membrane (or into the environment, if the rule is inside the $skin$ membrane); (a, in_j) says that a is sent into the child membrane with label j—if no such child membrane exists, the rule cannot be applied. If the special symbol δ appears in v, then the membrane which delimits the region is dissolved; in this way, all the objects in this region become elements of the surrounding membrane, while the rules of the dissolved membrane are removed. Since the skin is not allowed to be dissolved, we consider that the rules of the skin do not involve δ.

First we present an abstract syntax for membrane systems, and then a structural operational semantics of these systems by means of three sets of inference rules corresponding to maximal parallel rewriting, parallel communication, and parallel dissolving. A similar approach is presented in [2].

In general, operational semantics provide a way of rigorously describing the evolution of a computing system. Configurations are states of a system, and a computation consists of a sequence of transitions from one configuration to another, until a final configuration is reached.

Considering a set $\mathcal{R}$ of inference rules of the form $\frac{\text{premises}}{\text{conclusion}}$, the evolution of a membrane system can be presented as a deduction tree. A structural operational semantics of membrane systems emphasizes the deductive nature of membrane computing by describing the transition steps through a set of inference rules. A sequence of transition steps represents a *computation*. A computation is successful if this sequence is finite, namely there is no rule applicable to the objects present in the last committed configuration. In a halting committed configuration, the result of a successful computation is the total number of objects present either in the membrane considered as the output membrane, or in the outer region.

Let O be a finite alphabet of objects over which we consider the *free commutative monoid* O_c^*, whose elements are *multisets*. The empty multiset is denoted by *empty*. Objects can be enclosed in messages together with a target indication. We have *here* messages of typical form $(w, here)$, *out* messages (w, out), and *in* messages (w, in_L). For the sake of simplicity, hereinafter we consider that the messages with the same target indication merge into one message: $\prod_{i \in I}(v_i, here) = (w, here)$, $\prod_{i \in I}(v_i, in_L) = (w, in_L)$, $\prod_{i \in I}(v_i, out) = (w, out)$, with $w = \prod_{i \in I} v_i$, I a non-empty set, and $(v_i)_{i \in I}$ a family of multisets over O. In what follows the set $I = \{1, \ldots, n\}$ of the first n positive integers is denoted by $[n]$.

We use the mappings rules and priority to associate to a membrane label the set of evolution rules and the priority relation over rules (when this exists): $\mathsf{rules}(L_i) = R_i$, $\mathsf{priority}(L_i) = \rho_i$, and the projections L and w which return from a membrane its label and its current multiset, respectively.

The *set* $\mathcal{M}(\Pi)$ of membranes for a P system Π, and the *membrane structures* are defined inductively, as follows:

- if L is a label and w is a multiset over $O \cup (O \times \{here\}) \cup (O \times \{out\}) \cup \{\delta\}$, then $\langle L \mid w \rangle \in \mathcal{M}(\Pi)$; $\langle L \mid w \rangle$ is called a *simple (or elementary) membrane*, and it has the structure $\langle\rangle$;
- if L is a label, w is a multiset over $O \cup (O \times \{here\}) \cup (O \times \{in_{L(M_j)} \mid j \in [n]\}) \cup (O \times \{out\}) \cup \{\delta\}$, and M_1, …, $M_n \in \mathcal{M}(\Pi)$, $n \geq 1$, where each membrane M_i has the structure μ_i, then $\langle L \mid w ;\ M_1, \ldots, M_n \rangle \in \mathcal{M}(\Pi)$; $\langle L \mid w ;\ M_1, \ldots, M_n \rangle$ is called a composite membrane having the structure $\langle \mu_1, \ldots, \mu_n \rangle$.

We conventionally assume the existence of a set of sibling membranes denoted by $NULL$ such that M, $NULL = M = NULL, M$ and $\langle\ L \mid w\ ;\ NULL\ \rangle = \langle\ L \mid w\ \rangle$. The use of $NULL$ significantly simplifies several definitions and proofs. Let $\mathcal{M}^*(\Pi)$ be the free commutative monoid generated by $\mathcal{M}(\Pi)$ with the operation $(\_, \_)$ and the identity element $NULL$. We define $\mathcal{M}^+(\Pi)$ as the set of elements from $\mathcal{M}^*(\Pi)$ without the identity element. Let M_+, N_+ range over non-empty sets of sibling membranes, M_i over membranes, M_*, N_* range over possibly empty multisets of sibling membranes, and L over labels. The membranes preserve the initial labeling, evolution rules and priority relation among them in all subsequent configurations. Therefore, to describe a membrane we consider its label and the current multiset of objects together with its structure.

A *configuration* for a P system Π is a membrane structure together with the multisets of objects placed inside the membranes. Each membrane has no messages and no dissolving symbol δ, i.e., the multisets of all regions are elements in O_c^*. We denote by $\mathcal{C}(\Pi)$ the set of configurations for Π.

An *intermediate configuration* is a configuration in which we may find messages or the dissolving symbol δ. We denote by $\mathcal{C}^{\#}(\Pi)$ the set of intermediate configurations. We have $\mathcal{C}(\Pi) \subseteq \mathcal{C}^{\#}(\Pi)$.

Each membrane system Π has an initial configuration which is characterized by the initial multiset of objects for each membrane and the initial membrane structure of the system. For two configurations C_1 and C_2 of Π, we say that there is a *transition* from C_1 to C_2, and write $C_1 \Rightarrow C_2$, if the following *steps* are executed in the given order:

1. *maximal parallel rewriting step* each membrane evolves in a maximal parallel manner;
2. *parallel communication of objects through membranes* by sending and receiving messages;
3. *parallel membrane dissolving*, consisting in dissolving the membranes containing δ.

The last two steps take place only if there are messages and δ symbols resulting from the first step. If the first step is not possible, then neither are the other two steps; we say that the system has reached a *halting configuration*.

Maximal parallel rewriting step

We briefly present an operational semantics for membrane systems, considering each of the three steps. First we formally define the maximal parallel rewriting $\overset{mpr}{\Longrightarrow}_L$ for a multiset of objects in one membrane, and we extend it to maximal parallel rewriting $\overset{mpr}{\Longrightarrow}$ over several membranes. Some preliminary notions are required.

Definition 1 The irreducibility property w.r.t. the maximal parallel rewriting relation for multisets of objects, membranes, and for sets of sibling membranes is defined as follows:

- a multiset of messages and the dissolving symbol δ are **L-irreducible**;
- a multiset of objects w is **L-irreducible** iff there are no rules in $\mathsf{rules}(L)$ applicable to w with respect to the priority relation $\mathsf{priority}(L)$;
- a simple membrane $\langle\ L \mid w\ \rangle$ is **mpr-irreducible** iff w is L-irreducible;
- a non-empty set of sibling membranes $M_1, \ldots, M_n$ is **mpr-irreducible** iff M_i is mpr-irreducible for every $i \in [n]$; $NULL$ is **mpr-irreducible**;
- a composite membrane $\langle\ L \mid w\ ;\ M_1, \ldots, M_n\ \rangle$ is **mpr-irreducible** iff w is L-irreducible, and the set of sibling membranes $M_1, \ldots, M_n$ is mpr-irreducible.

The priority relation is a form of control on the application of rules. In the presence of a priority relation, no rule of a lower priority can be used during the same evolution step when a rule with a higher priority is used, even if the two rules do not compete for the same objects. We formalize the conditions imposed by the priority relation on rule applications in the definition below.

Definition 2 Let M be a membrane labeled by L, and w a multiset of objects. A non-empty multiset $R = (u_1 \to v_1, \ldots, u_n \to v_n)$ of evolution rules is (L, w)-**consistent** if:

- R is a multiset of rules from $\mathsf{rules}(L)$,
- $w = u_1 \ldots u_n z$, so each rule $r \in R$ is applicable on w,
- $(\forall r \in R, \forall r' \in \mathsf{rules}(L))$ r' applicable on w implies $(r', r) \notin \mathsf{priority}(L)$ (we have $(r_1, r_2) \in \mathsf{priority}(L)$ iff $r_1 > r_2$),
- $(\forall r', r'' \in R)$ $(r', r'') \notin \mathsf{priority}(L)$,
- the dissolving symbol δ has at most one occurrence in the multiset $v_1 \ldots v_n$.

Maximal parallel rewriting relations $\overset{mpr}{\Longrightarrow}_L$ and $\overset{mpr}{\Longrightarrow}$ are defined by the following inference rules:

For each $w = u_1 \ldots u_n z \in O_c^+$ such that z is L-irreducible, and (L, w)-consistent rules $(u_1 \to v_1, \ldots, u_n \to v_n)$,

$$(\mathbf{R_1})\ \frac{}{u_1 \ldots u_n z \overset{mpr}{\Longrightarrow}_L v_1 \ldots v_n z}$$

For each $w \in O_c^+$, $w' \in (O \cup Msg(O) \cup \{\delta\})_c^*$, and mpr-irreducible $M_* \in \mathcal{M}^*(\Pi)$,

$$(\mathbf{R_2}) \ \frac{w \overset{mpr}{\Longrightarrow}_L w'}{\langle\, L \mid w\ ;\ M_*\,\rangle \overset{mpr}{\Longrightarrow} \langle\, L \mid w'\ ;\ M_*\,\rangle}$$

For each L-irreducible $w \in O_c^*$, and $M_+, M'_+ \in \mathcal{M}^+(\Pi)$,

$$(\mathbf{R_3}) \ \frac{M_+ \overset{mpr}{\Longrightarrow} M'_+}{\langle\, L \mid w\ ;\ M_+\,\rangle \overset{mpr}{\Longrightarrow} \langle\, L \mid w\ ;\ M'_+\,\rangle}$$

For each $w \in O_c^+$, $w' \in (O \cup Msg(O) \cup \{\delta\})_c^+$, $M_+, M'_+ \in \mathcal{M}^+(\Pi)$,

$$(\mathbf{R_4}) \ \frac{w \overset{mpr}{\Longrightarrow}_L w', M_+ \overset{mpr}{\Longrightarrow} M'_+}{\langle\, L \mid w\ ;\ M_+\,\rangle \overset{mpr}{\Longrightarrow} \langle\, L \mid w'\ ;\ M'_+\,\rangle}$$

For each $M, M' \in \mathcal{M}(\Pi)$, and $M_+, M'_+ \in \mathcal{M}^+(\Pi)$,

$$(\mathbf{R_5}) \ \frac{M \overset{mpr}{\Longrightarrow} M', M_+ \overset{mpr}{\Longrightarrow} M'_+}{M, M_+ \overset{mpr}{\Longrightarrow} M', M'_+}$$

For each $M, M' \in \mathcal{M}(\Pi)$, and mpr-irreducible $M_+ \in \mathcal{M}^+(\Pi)$,

$$(\mathbf{R_6}) \ \frac{M \overset{mpr}{\Longrightarrow} M'}{M, M_+ \overset{mpr}{\Longrightarrow} M', M_+}$$

We note that $\overset{mpr}{\Longrightarrow}$ for simple membranes can be described by rule $(\mathbf{R_2})$ with $M_* = NULL$.

Remark 1 M is mpr-irreducible iff it does not exist M' such that $M \overset{mpr}{\Longrightarrow} M'$.

Proposition 1 *Let Π be a membrane system. If $C \in \mathcal{C}(\Pi)$ and $C' \in \mathcal{C}^{\#}(\Pi)$ such that $C \overset{mpr}{\Longrightarrow} C'$, then C' is mpr-irreducible.*

The proof follows by structural induction on C.

The formal definition of $\overset{mpr}{\Longrightarrow}$ given above corresponds to the intuitive description of maximal parallelism. The nondeterminism is given by the associativity and commutativity of the concatenation operation over objects used in $\mathbf{R}_1$. The parallelism of the evolution rules in a membrane is also given by $\mathbf{R}_1 : u_1 \ldots u_n z \overset{mpr}{\Longrightarrow}_L v_1 \ldots v_n z$ says that the rules of themultiset $(u_1 \to v_1, \ldots, u_n \to v_n)$ are applied simultaneously. The fact that the membranes evolve in parallel is described by rules $\mathbf{R_3} - \mathbf{R_6}$.

Parallel communication among membranes

We say that a multiset w is *here-free/out-free/ in_L-free* if it does not contain any *here/out/ in_L* messages, respectively. For w a multiset of objects and messages, we introduce the operations obj, here, out, and in_L as follows:

$\mathsf{obj}(w)$ is obtained from w by removing all messages,

$$\mathsf{here}(w) = \begin{cases} empty & \text{if } w \text{ is } here\text{-free}, \\ w'' & \text{if } w = w'(w'', here) \wedge w' \text{ is } here\text{-free}; \end{cases}$$

$$\mathsf{out}(w) = \begin{cases} empty & \text{if } w \text{ is } out\text{-free}, \\ w'' & \text{if } w = w'(w'', out) \wedge w' \text{ is } out\text{-free}; \end{cases}$$

$$\mathsf{in}_L(w) = \begin{cases} empty & \text{if } w \text{ is } in_L\text{-free}, \\ w'' & \text{if } w = w'(w'', in_L) \wedge w' \text{ is } in_L\text{-free}. \end{cases}$$

We consider the extension of the operator w (previously defined over membranes) to non-empty sets of sibling membranes by setting $\mathsf{w}(NULL) = empty$ and $\mathsf{w}(M_1, \ldots, M_n) = \mathsf{w}(M_1) \ldots \mathsf{w}(M_n)$.

We recall that the messages with the same target merge in one larger message.

Definition 3 The **tar-irreducibility** property for membranes and for sets of sibling membranes is defined as follows:

- a simple membrane $\langle L|w\rangle$ is **tar-irreducible** iff w is $here$-free and $L \neq Skin \vee (L = Skin \wedge wout - \text{free})$;
- a non-empty set of sibling membranes $M_1, \ldots, M_n$ is **tar-irreducible** iff M_i is tar -irreducible for every $i \in [n]$; $NULL$ is **tar-irreducible**;
- a composite membrane $\langle\, L \mid w\ ;\ M_1, \ldots, M_n\,\rangle, n \geq 1$, is **tar-irreducible** iff: w is $here$-free and $in_{L(M_i)}$-free for every $i \in [n]$, $L \neq Skin \vee (L = Skin \wedge w$ is out-free), $\mathsf{w}(M_i)$ is out-free for all $i \in [n]$, and the set of sibling membranes $M_1, \ldots, M_n$ is tar-irreducible;

Notation We treat messages of the form $(w', here)$ as a particular communication inside a membrane, and we substitute $(w', here)$ by w'. We denote by $\overline{w}$ the multiset obtained by replacing $(\mathsf{here}(w), here)$ with $\mathsf{here}(w)$ in w. For instance, if $w = a\,(bc, here)\,(d, out)$ then $\overline{w} = abc\,(d, out)$, where $\mathsf{here}(w) = bc$. We note that $\mathsf{in}_L(\overline{w}) = \mathsf{in}_L(w)$, and $\mathsf{out}(\overline{w}) = \mathsf{out}(w)$.

The **parallel communication relation** $\overset{tar}{\Longrightarrow}$ is defined by the following inference rules:

For each tar-irreducible $M_* \in \mathcal{M}^*(\Pi)$ and multiset w such that $\mathsf{here}(w) \neq empty$, or $L = Skin \wedge \mathsf{out}(w) \neq empty$, or there exists $M_i \in M_*$ with $\mathsf{in}_{L(M_i)}(w)\mathsf{out}(\mathsf{w}(M_i)) \neq empty$,

$$(\mathbf{C_1})\ \frac{}{\langle\, L \mid w\ ;\ M_*\,\rangle \overset{tar}{\Longrightarrow} \langle\, L \mid w'\ ;\ M'_*\,\rangle}$$

where

$$w' = \begin{cases} \mathsf{obj}(\overline{w})\ \mathsf{out}(\mathsf{w}(M_*)) & \text{if } L = Skin, \\ \mathsf{obj}(\overline{w})\ (\mathsf{out}(w), out)\ \mathsf{out}(\mathsf{w}(M_*)) & \text{otherwise;} \end{cases}$$

and

$$\mathsf{w}(M'_i) = \mathsf{obj}(\mathsf{w}(M'_i))\ \mathsf{in}_{\mathsf{L}(M_i)}(w),\ \text{for all } M_i \in M_*$$

For each $M_1, \ldots, M_n, M'_1, \ldots, M'_n \in \mathcal{M}^+(\Pi)$, and each w,

$$(\mathbf{C_2})\ \frac{M_1, \ldots, M_n \overset{tar}{\Longrightarrow} M'_1, \ldots, M'_n}{\langle\, L \mid w\ ;\ M_1, \ldots, M_n\,\rangle \overset{tar}{\Longrightarrow} \langle\, L \mid w''\ ;\ M''_1, \ldots, M''_n\,\rangle}$$

where

$$w'' = \begin{cases} \mathsf{obj}(\overline{w})\ \mathsf{out}(\mathsf{w}(M'_1, \ldots, M'_n)) & \text{if } L = Skin, \\ \mathsf{obj}(\overline{w})\ (\mathsf{out}(w), out)\ \mathsf{out}(\mathsf{w}(M'_1, \ldots, M'_n)) & \\ \text{otherwise;} & \end{cases}$$

and each M''_i is obtained from M'_i by replacing its resources with

$$\mathsf{w}(M''_i) = \mathsf{obj}(\overline{\mathsf{w}(M'_i)})\ \mathsf{in}_{\mathsf{L}(M'_i)}(w),\ \text{for all } i \in [n]$$

For each $M, M' \in \mathcal{M}(\Pi)$, and tar-irreducible $M_+ \in \mathcal{M}^+(\Pi)$,

$$(\mathbf{C_3})\ \frac{M \overset{tar}{\Longrightarrow} M'}{M, M_+ \overset{tar}{\Longrightarrow} M', M_+}$$

For each $M \in \mathcal{M}(\Pi)$, $M_+ \in \mathcal{M}^+(\Pi)$,

$$(\mathbf{C_4})\ \frac{M \overset{tar}{\Longrightarrow} M', M_+ \overset{tar}{\Longrightarrow} M'_+}{M, M_+ \overset{tar}{\Longrightarrow} M', M'_+}$$

Remark 2 M is tar-irreducible iff there does not exist M' such that $M \overset{tar}{\Longrightarrow} M'$.

Proposition 2 *Let Π be a membrane system. If $C \in \mathcal{C}^{\#}(\Pi)$ with messages and $C \overset{tar}{\Longrightarrow} C'$, then C' is tar-irreducible.*

Parallel membrane dissolving

If the special symbol δ occurs in the multiset of objects of a membrane labeled by L, that membrane is dissolved, its evolution rules and the associated priority relation are lost, and its contents (objects and membranes) are added to the contents of the surrounding membrane. We say that a multiset w is δ-*free* if it does not contain the special symbol δ.

Definition 4 The **δ-irreducibility** property for membranes and for sets of sibling membranes is defined as follows:

- a simple membrane is **δ-irreducible** iff it has no messages and is δ-free;
- a non-empty set of sibling membranes $M_1, \ldots, M_n$ is **δ-irreducible** iff every membrane M_i is δ-irreducible, for $1 \le i \le n$; $NULL$ is **δ-irreducible**;
- a composite membrane $\langle\, L \mid w\ ;\ M_+\,\rangle$ is **δ-irreducible** iff w has no messages, M_+ is δ-irreducible, and $w(M_+)$ is δ-free.

Parallel dissolving relation $\overset{\delta}{\Longrightarrow}$ is defined by the following inference rules:

For each δ-irreducible $M_* \in \mathcal{M}^*(\Pi)$, $\langle\, L_2 \mid w_2\delta\ ;\ M_*\,\rangle$, and label L_1,

$$(\mathbf{D_1})\ \frac{}{\langle\, L_1 \mid w_1\ ;\ \langle\, L_2 \mid w_2\delta\ ;\ M_*\,\rangle\,\rangle \overset{\delta}{\Longrightarrow} \langle\, L_1 \mid w_1 w_2\ ;\ M_*\,\rangle}$$

For each $M_+ \in \mathcal{M}^+(\Pi)$, $M'_* \in \mathcal{M}^*(\Pi)$, δ-free multiset w_2, multisets w_1, w'_2, and labels L_1, L_2

$$(\mathbf{D_2})\ \frac{\langle\, L_2 \mid w_2\ ;\ M_+\,\rangle \overset{\delta}{\Longrightarrow} \langle\, L_2 \mid w'_2\ ;\ M'_*\,\rangle}{\langle\, L_1 \mid w_1\ ;\ \langle\, L_2 \mid w_2\ ;\ M_+\,\rangle\,\rangle \overset{\delta}{\Longrightarrow} \langle\, L_1 \mid w_1\ ;\ \langle\, L_2 \mid w'_2\ ;\ M'_*\,\rangle\,\rangle}$$

For each $M_+ \in \mathcal{M}^+(\Pi)$, $M'_* \in \mathcal{M}^*(\Pi)$, multisets w_1, w_2, w'_2, and labels L_1, L_2

$$(\mathbf{D_3})\ \frac{\langle\, L_2 \mid w_2\delta\ ;\ M_+\,\rangle \overset{\delta}{\Longrightarrow} \langle\, L_2 \mid w'_2\delta\ ;\ M'_*\,\rangle}{\langle\, L_1 \mid w_1\ ;\ \langle\, L_2 \mid w_2\delta\ ;\ M_+\,\rangle\,\rangle \overset{\delta}{\Longrightarrow} \langle\, L_1 \mid w_1 w'_2\ ;\ M'_*\,\rangle}$$

For each $M_+ \in \mathcal{M}^+(\Pi)$, $M'_*, N'_* \in \mathcal{M}^*(\Pi)$, δ-irreducible $\langle\, L \mid w\ ;\ N_+\,\rangle$, and multisets w', w'',

$$(\mathbf{D_4})\ \frac{\langle\, L \mid w\ ;\ M_+\,\rangle \overset{\delta}{\Longrightarrow} \langle\, L \mid w'\ ;\ M'_*\,\rangle}{\langle\, L \mid w\ ;\ M_+, N_+\,\rangle \overset{\delta}{\Longrightarrow} \langle\, L \mid w'\ ;\ M'_*, N_+\,\rangle}$$

$$(\mathbf{D_5})\ \frac{\langle\, L \mid w\ ;\ M_+ \,\rangle \overset{\delta}{\Longrightarrow} \langle\, L \mid ww'\ ;\ M'_* \,\rangle \quad \langle\, L \mid w\ ;\ N_+ \,\rangle \overset{\delta}{\Longrightarrow} \langle\, L \mid ww''\ ;\ N'_* \,\rangle}{\langle\, L \mid w\ ;\ M_+, N_+ \,\rangle \overset{\delta}{\Longrightarrow} \langle\, L \mid ww'w''\ ;\ M'_*, N'_* \,\rangle}$$

Remark 3 M is δ-irreducible iff there does not exist M' such that $M \overset{\delta}{\Longrightarrow} M'$.

Proposition 3 *Let* Π *be a membrane system. If* $C \in \mathcal{C}^{\#}(\Pi)$ *is tar-irreducible and* $C \overset{\delta}{\Longrightarrow} C'$*, then* C' *is* δ*-irreducible.*

It is worth noting that $C \in \mathcal{C}(\Pi)$ iff C is tar-irreducible and δ-irreducible. According to the standard description in membrane computing, a *transition step* between two configurations $C, C' \in \mathcal{C}(\Pi)$ is given by: $C \Rightarrow C'$ iff C and C' are related by one of the following relations:

either $C \overset{mpr}{\Longrightarrow}; \overset{tar}{\Longrightarrow} C'$,
or $C \overset{mpr}{\Longrightarrow}; \overset{\delta}{\Longrightarrow} C'$,
or $C \overset{mpr}{\Longrightarrow}; \overset{tar}{\Longrightarrow}; \overset{\delta}{\Longrightarrow} C'$.

The three alternatives in defining $C \Rightarrow C'$ are given by the existence of messages and dissolving symbols along the system evolution. Starting from a configuration without messages and dissolving symbols, we apply the "mpr" rules and get an intermediate configuration which is mpr-irreducible; if we have messages, then we apply the "tar" rules and get an intermediate configuration which is tar-irreducible; if we have dissolving symbols, then we apply the dissolving rules and get a configuration which is δ-irreducible. After applying "mpr"-step there are either messages, δ symbols or both. If we have messages (could be only *here* messages) we should apply the"tar" step; otherwise, for δ objects we should apply δ step. If the last configuration has no messages or dissolving symbols, then we say that the transition relation $\Rightarrow$ is well defined as an evolution step between the first and last configurations.

Proposition 4 *The relation* $\Rightarrow$ *is well defined over the entire set* $\mathcal{C}(\Pi)$ *of configurations.*

Examples of inference trees are presented in [2].

Synchronization issues in membrane systems

It is evident from the operational semantics that there are several synchronization aspects related to the evolution of a membrane system.

The relationship between the synchronous and the asynchronous approaches in computing systems, particularly in massively parallel and multiprocessor computing systems, will remain a challenging topic for many years to come. There are reasons to think that the asynchronous approach has some advantages; however, the synchronous methodology prevails in the modern computing systems architecture. As if this is not enough, different fields treat the concepts of synchrony and asynchrony somewhat differently. The main terms (parallelism, concurrency, time) should be clarified to discuss the synchronous and asynchronous issues. In our approach we work with a "causal" time (defined as the partial order on some events resulting from their cause–effect relationships) rather a physical time (defined as an independent physical variable related to a clock). The concept of causal time was formulated initially by Aristotle (If nothing happens, no time); it can be useful in systems dealing with events defining cause–effect relationships. The abstract model of a finite state machine corresponds to the model of an asynchronous system evolving in logical time; a possible conversion to a synchronous approach is given by a barrier synchronization (as an engineering solution) to manage unpredictable variations of the delays introduced by real physical components. An algorithm (its program) consists of a sequence of steps which perform some actions. Asynchrony is usually treated as the dependence of the number of steps required to obtain the result on the input data. In the case of a fully sequential algorithm (program), such treatment of asynchrony is important only for performance evaluation. Parallel algorithms and programs present new and challenging tasks. Certain steps of an algorithm can be performed concurrently. Representing an algorithm (program) in the form suitable for concurrent implementation is reduced to the cause–effect relationships between the operations (processes, commands) in the algorithm. Thus, a parallel specification is a procedure for introducing logical time into the algorithm. An implementation of a global synchronous system can be given by delivering a termination signal from the processors (processes) of the system. Difficulties appear when several processes have a shared resource, and non-synchronized events may occur. A possible solution of a synchronous implementation that eliminates the problems of physical asynchrony is as follows:

- every process can be in two phases: active and passive;
- a process can run only when active;
- to transit from passive to active a process has to receive a signal;

– after an active process executes, it signals other passive processes;

Initially we activate some processes, which after their executions signal passive processes. This repeats until all processes have terminated. Following this scenario, deadlock can occur if the process dependency graph contains cycles. In this scenario, process can be synchronized using a barrier. A *process barrier* is an concurrent abstraction through which multiple processes can be synchronized. Thus, a passive process can be considered a process that is waiting at the barrier, and by passing the barrier it becomes an active one.

We can apply this type of synchronization to membrane systems, by allowing a membrane to evolve only after it has passed the barrier. To model this, we use a set of antecedents and a set of descendants for each membrane when describing the system. To apply its rules, a membrane needs to receive signals from all of its antecedents. After it applies its rules, the membrane signals all of its descendants. The set of antecedents specifies how many times a signal needs to be received by each membrane. The set of descendants specifies the membranes that need to be signaled after the application of rules.

Using this mechanism, we can control the relative evolution speed of the antecedents of a membrane. This approach allows to specify that a certain membrane can repeat its step several times before sending its signal to the descendents. In this way we can have a *parameterized synchronization* between membranes, and this aspect could be very useful in modeling biological phenomena. The evolution of a membrane can be described by the following steps which are repeated until no rule can be applied:

1. collect signals from all the antecedents;
2. apply the rules after receiving all the signals;
3. signal all descendants.

A scalable implementation of membrane systems

We have selected GridGain [12] as our platform because it provides all the required features, and it is easily deployed on multiple platforms. GridGain systems develop (open source) cloud applications that facilitate the development of highly scalable applications that work natively on any managed infrastructure (from a single Android device to large grids or clouds). GridGain software supports all major operating systems and provides native support for Java and Scala programming languages.

Using GridGain, we present a highly scalable distributed implementation of membrane systems in which we emphasize the notion of computation and synchronization. Distributed computations with GridGain are performed in parallel fashion, gaining high performance and low latency. GridGain allows the user to distribute computations and data processing across multiple computers in a cluster, a grid or a cloud. Distributed parallel processing is based on the ability of executing any computation on any set of cluster nodes. To achieve scalability we make use of *MapReduce*. The paradigm is defined by two main steps: *map* and *reduce*. The map step allows splitting a task into multiple jobs that execute in parallel on the nodes. The reduce step aggregates the result of each job and returns the task result.

GridGain is a Java-based open source computing infrastructure released under LGPL license. It provides a zero deployment model, meaning that a node can be deployed by running a script, or by creating a node instance. A valuable feature of the system is its support for advanced load balancing and scheduling by providing early and late load balancing that are defined by load balancing and collision (scheduling) resolution. Another important feature is pluggable fault tolerance with several popular implementations available out of the box. It allows the failover of logic and not only the data. The most notable features of GridGain we use are: tasks and jobs modeled according to the MapReduce paradigm, communication between tasks and jobs, as well as on-demand class loading.

The simulation of a membrane system can be viewed as a task. The jobs associated with this task define the execution of each membrane. Hence, the number of jobs is equal to the number of membranes. To model the proposed synchronization mechanism between membranes, a communication between jobs is required. We employ a synchronization mechanism based on certain preconditions expressing the consistency of the global state of the system. This synchronization mechanism has been introduced to control the dependency relation between membranes. We propose a synchronous model of execution used to coordinate membrane evolution.

The main steps of the simulation are: (1) build a membrane system from an specification file; (2) using the generated membrane system, construct and execute a grid job: (i) *Map*: create a job for each membrane; (ii) *reduce*: gather all the responses from the jobs and create the resulting membrane system. The simulation repeats step 2 as long as a rule is applied. Each generated job contains an object that describes a membrane from the system. The job is responsible for the correct simulation of the evolution of the membrane. Thus, it needs to synchronize with other membranes, and also to

Fig. 1 Membrane class

```
public class Membrane {
  private List<MembraneLabel> childrenLabels;
  private List<Rule> rules;
  private HashMultiset contents;
  private HashMultiset incomingObjects;
  private MembraneLabel label;
  private MembraneLabel parentLabel;
  private HashMap<MembraneLabel, Integer> antecedents;
  private List<MembraneLabel> descendants;
  private int appliedRules;//number of applied rules in this step

  public Membrane()
  //test if the membrane contains a multiset
  public boolean contains(HashMultiset multiset)
  //store a  multiset that resulted in this evolution step
  public void enqueMultiset(HashMultiset multiset)
  //add the objects that resulted in this evolution step
  public void endEvolution()
  //return the list of applicable rules
  public List<Rule> getApplicableRules()
}
```

Fig. 2 Rule class

```
public class Rule extends RuleConstraint {
  List<RuleConstraint> constraints;

  public Rule()
  public void apply(Membrane membrane) {
    for (RuleConstraint constraint : constraints) {
      constraint.apply(membrane);
    }
  }
  public boolean check(Membrane membrane) {
    boolean isApplicable = true;
    Iterator<RuleConstraint> iter = constraints.iterator();
    while (isApplicable && iter.hasNext()) {
      isApplicable = isApplicable && iter.next().check(membrane);
    }
    return isApplicable;
  }
}
```

apply different rules. The result of the job consists of the final state of the simulated membrane.

We have used a modular design for the entities of the system in which we separated the objects defining the *grid behavior* from those defining the *membrane systems*. Thus, we implement several abstractions that model various notions such as: *membranes*, *rules*, membrane *objects*, etc. For the grid behavior we define the following concepts: *task*, *job*, *barrier*.

In Fig. 1 we describe the members and main methods of class Membrane. The object is responsible only for operations that modify the contents of a membrane. The evolution logic is implemented using the Rule and EvolutionVisitor objects. To model the rules of a membrane system we used an extensible approach. Each rule can be seen as a list of constraints; a constraint is responsible for checking if its precondition is valid (via method *check*), and for applying its postcondition on a membrane (via method *apply*).

The main methods of the Rule class are presented in Fig. 2. Using these abstractions we can implement rules with various ingredients, only by describing constraints and aggregating them into a new type of Rule. The evolution of a membrane is performed by the EvolutionVisitor class. The method *localMembraneEvolution* defines the logic of a single step of evolution. A step is simulated by the repeated application of rules.

A grid task is defined by the class PsTask (Fig. 3), which follows the MapReduce paradigm. The method *split* takes as input a membrane system, and for each membrane creates a job that will be executed on the grid. The method *reduce* receives a list of job results that

Fig. 3 PsTask class

```
public class PsTask{
  public MembraneSystem reduce(List results){
    MembraneSystem result = new MembraneSystem();
    int appliedRules = 0;
    for (GridJobResult gridJobResult : results) {
      Membrane data = gridJobResult.getData();
      result.getMembranes().put(data.getLabel(), data);
      appliedRules += data.getAppliedRules();
    }
    result.setAppliedRules(appliedRules);
    return result;
  }
  protected List<GridJob> split(MembraneSystem arg){
    List<PsJob> jobs = new ArrayList<PsJob>();
    for(Membrane mbr : arg.getMembranes().values()){
      jobs.add(new PsJob(mbr));
    }
    return jobs;
  }
}
```

contain membranes, and assembles them in a membrane system.

A grid job is described by the PsJob object. This object contains a membrane which holds the data, and a barrier used for synchronization. The main method of this class is *execute*, in which the evolution of a membrane is executed. The evolution consists of a three-step loop: (i) wait at the barrier for incoming signals, (ii) after receiving the signals, apply the rules, and (iii) after applying the rules, signal the descendants. The result of the job is a maximally parallel step of the membrane.

Membrane synchronization is achieved using a special form of barrier. The barrier waits to be signaled from each antecedent membrane a specified number of times. After this, it releases the job that called the method *waitAt*. The barrier also listens for termination signals. When it receives such a signal it informs the waiting job that it should finish its execution.

Example

We provide a simple example to illustrate the simulator. The system is composed of two membranes. Membrane $m1$ contains a^{2000} and has rules $a \to b$, and $b^2 \to d$, while membrane $m2$ contains $a^{40000}b^{1000}c^{5000}$ and has rules $a^2 \to b$, and $c^2 \to d$. The signaling part is denoted by the contents of *wait*, and *signal*. Those include a sequence of membranes and the number of times they have to signal. Notice that $m2$ has to wait to be signaled by $m1$ two times before it can apply a rule. The parent of $m2$ is $m1$, which is the skin membrane.

```
/* PsGrid input file */
membrane m1 /*name of the membrane*/ :
      skin /*name of the parent*/{
  children {
    m1  /*name of children*/
  }
  contents {
    a^{2000}/*contents of the membrane*/
  }
  rules {
    /*rules of the membrane*/
    [a^{1} ==> b^{1}]
    [b^{2} ==> d^{1}]
  }
  wait{
    /*the antecedents*/
  }
  signal{
    m2 /*the descendants*/
  }
}
membrane m2 : m1{
  children {
  }
  contents {
    a^{40000}b^{1000}c^{5000}
  }
  rules {
    [a^{2} ==> b^{1}]
    [c^{2} ==> d^{1}]
  }
  wait{
    m1^{2}
  }
  signal{
  }
}
```

We also present the log from each node of the grid. The log shows the order in which membrane jobs arrive at each node, and the actions they execute. The number of rule applications executed in a certain step is written at the end of the lines (after #). Notice that the job ends if it receives a terminate signal, or if the membrane did not apply any rules in this step.

```
[20:26:56,843][INFO ][gridgain-#6%null%][PsJob] Received membrane with
     contents :m1:[a^{2000} ] [[Rule:  in m1[a  -> b ], Rule:  in m1[b
    ^{2}  -> d ]]] #0
[20:26:56,843][INFO ][gridgain-#10%null%][PsJob] Received membrane
    with contents :m2:[b^{1000} c^{5000} a^{40000} ] [[Rule:  in m2[a
    ^{2}  -> b ], Rule:  in m2[c^{2}  -> d ]]] #0
[20:26:56,843][INFO ][gridgain-#10%null%][PsJob] Waiting at barrier:m2
[20:26:56,843][INFO ][gridgain-#6%null%][PsJob] Waiting at barrier:m1
[20:26:56,843][INFO ][gridgain-#6%null%][PsJob] Passing the barrier:m1
[20:26:56,875][INFO ][gridgain-#6%null%][PsJob] Sending signal to
    descendants
[20:26:56,890][INFO ][gridgain-#6%null%][PsJob] After evolution :m1:[b
    ^{2000} ] [[Rule:  in m1[a  -> b ], Rule:  in m1[b^{2}  -> d ]]]
    #2000
[20:26:56,890][INFO ][gridgain-#6%null%][PsJob] Waiting at barrier:m1
[20:26:56,890][INFO ][gridgain-#6%null%][PsJob] Passing the barrier:m1
[20:26:56,890][INFO ][gridgain-#10%null%][PsJob] Passing the barrier:
    m2
[20:26:56,906][INFO ][gridgain-#6%null%][PsJob] Sending signal to
    descendants
[20:26:56,921][INFO ][gridgain-#6%null%][PsJob] After evolution :m1:[d
    ^{1000} ] [[Rule:  in m1[a  -> b ], Rule:  in m1[b^{2}  -> d ]]]
    #1000
[20:26:56,921][INFO ][gridgain-#6%null%][PsJob] Waiting at barrier:m1
[20:26:56,921][INFO ][gridgain-#6%null%][PsJob] Passing the barrier:m1
[20:26:56,921][INFO ][gridgain-#6%null%][PsJob] Sending signal to
    descendants
[20:26:56,921][INFO ][gridgain-#6%null%][PsJob] After evolution :m1:[d
    ^{1000} ] [[Rule:  in m1[a  -> b ], Rule:  in m1[b^{2}  -> d ]]]
    #0
[20:26:57,062][INFO ][gridgain-#10%null%][PsJob] Sending signal to
    descendants
[20:26:57,062][INFO ][gridgain-#10%null%][PsJob] After evolution :m2:[
    d^{2500} b^{21000} ] [[Rule:  in m2[a^{2}  -> b ], Rule:  in m2[c
    ^{2}  -> d ]]] #22500
[20:26:57,062][INFO ][gridgain-#10%null%][PsJob] Waiting at barrier:m2
[20:26:57,062][INFO ][gridgain-#10%null%][PsJob] Passing the barrier:
    m2
[20:26:57,062][INFO ][gridgain-#10%null%][PsJob] Sending signal to
    descendants
[20:26:57,062][INFO ][gridgain-#10%null%][PsJob] After evolution :m2:[
    d^{2500} b^{21000} ] [[Rule:  in m2[a^{2}  -> b ], Rule:  in m2[c
    ^{2}  -> d ]]] #0
```

The simulator has a simple but flexible graphical interface. A screenshot after executing a simulation is presented in Fig. 4. The first row presents the initial configuration of the membrane system. The second row presents the contents of the membranes after the simulation.

Even though this example is simple, the implementation can benefit from several features of GridGain and provide a complex parallel implementation of membrane systems. The main points are that the implementation is faithful to the formal description of the membrane systems, and it is also scalable to a high number of membranes (which is the case in cell biology simulations).

Conclusion

Hierarchies are often used in modeling and simulation for computational biology. A hierarchical perspective of the cell considers components structured into classes of similar kinds, e.g. golgi, ER, and nucleus form organelles, i.e. membrane-bound compartments of the cell. New models of membrane systems need to be simulated on complex hardware systems to provide a valuable feedback to biologists. Membrane computing is a branch of natural computing using an explicit hierarchical description coming exactly from the structure and functioning of the living cell. The main areas where membrane computing has been used as a modeling framework (biology and bio-medicine, linguistics, economics, computer science, etc.) are presented in [7]. In that volume, several implementations (mainly using sequential computational environments) for simulating various types of cell-like membrane systems are presented in [8]. We consider the simulation of membrane systems using sequential computers as inappropriate, because membrane systems are intrinsically parallel and nondeterministic computational devices and their computation trees are difficult to store and handle with one processor. Therefore, it is necessary to look for parallel and scalable implementations able to simulate as close as possible the formal description of the membrane systems.

In this paper we present a faithful parallel implementation of membrane systems using GridGain, emphasizing on the synchronization problems appearing in membrane computing. Thus we hope to offer a suitable simulator for

Fig. 4 PsGrid screen after executing the example

membrane systems, opening a new possibility of using membrane computing as a parallel and nondeterministic modeling framework for addressing structural and dynamical aspects of complex systems modeling phenomena in cell biology where huge number of elements are used.

In the papers devoted to membrane systems it is not mentioned how the membranes (or groups of membranes) interact or synchronize. The usual thinking is that membrane systems are synchronized locally (a step of a membrane is given by the parallel application of rules) and behave asynchronously at the global level. We emphasize here the global aspects, by adding a form of parameterized barrier synchronization between membranes. A parallel implementation of membrane systems is presented in [6]. It uses a cluster of 64 dual processors, and an MPI library to describe the communication and synchronization of parallel processes. In that parallel simulator, the rules are implemented as threads. At the system initialization phase, one thread is created for each rule. Within one membrane, several rules can be applied concurrently. This parallelism between rule applications within one membrane is modeled with multithreading. Rule applications are performed in terms of rounds. To synchronize each thread (rule) within the system, two barriers implemented as mutexes are associated with a thread. At the beginning of each round, the barrier that the rule thread is waiting on is released by the primary controlling thread. After the rule application is done, the thread waits for the second barrier, and the primary thread locks the first barrier. During the following round it would repeat the above procedure, releasing and locking alternating barriers. Since many rules are executing concurrently and they are sharing resources, a mutual exclusion algorithm is necessary. The communication and synchronization between membranes are implemented using the Message Passing Interface library of functions for parallel computation. The execution is performed in terms of rounds; at the end of each round, every membrane exchanges messages with all its children and parent before proceeding to the next round. Another concern is the termination detection problem.

Recently several simulators have been produced to model the behavior of various classes of membrane systems, including a web-based one [3]. An executable specification for P systems [1] is implemented in Maude, a software system supporting rewriting and equational logic. A parallel simulation of P systems has been done using GPU in [9], while in [4] is proposed a simulation of active membranes using CUDA architectures. Some fundamental distributed algorithms applied in this special framework and used in these implementations of membrane systems are presented in [5].

Acknowledgments Many thanks for the helpful comments and discussions to Bogdan Aman and anonymous reviewer. This work was partially supported by a grant of the Romanian Authority for Scientific Research, project number PN-II-ID-PCE-2011-3-0919.

References

1. Andrei O, Ciobanu G, Lucanu D (2005) Executable Specifications of P Systems. Lect Notes Comput Sci 3365:126–145
2. Andrei O, Ciobanu G, Lucanu D (2007) A rewriting logic framework for operational semantics of membrane systems. Theor Comput Sci 373:163–181
3. Bonchiş C, Ciobanu G, Izbaşa C, Petcu D (2005) A web-based P Systems simulator and its parallelization. Lect Notes Comput Sci 3699:58–69
4. Cecilia JM, Garcia JM, Guerrero GD, Martinez-del-Amor MA, Pérez-Hurtado I, Pérez-Jiménez MJ (2010) Simulation of P Systems with Active Membranes on CUDA. Brief Bioinf 11:313–322
5. Ciobanu G (2003) Distributed algorithms over communicating membrane systems. Biosystems vol. 70, Elsevier, pp 123–133
6. Ciobanu G, Guo W (2004) P Systems Running on a Cluster of Computers. Membrane Computing. Lect Notes Comput Sci. vol. 2933, Springer, pp 123–139
7. Ciobanu G, Păun Gh, Pérez-Jiménez MJ (eds) (2006) Applications of Membrane Computing. Natural Computing Series, Springer
8. Gutiérrez-Naranjo MA, Pérez-Jiménez MJ, Riscos-Núñez A (2006) Available Membrane Computing Software. In: Ciobanu G, Păun Gh, Pérez-Jiménez MJ (eds) Applications of Membrane Computing, vol 7, Springer, pp 411–436
9. Martinez-del-Amor MA, Perez-Carrasco J, Pérez-Jiménez MJ (2013) Characterizing the parallel simulation of P systems on the GPU. Int J Unconv Comput 9:405–424
10. Păun Gh (2002) Membrane Computing. An Introduction, Springer
11. Păun Gh, Rozenberg G, Salomaa A (2010) The Oxford Handbook of Membrane Computing, Oxford University Press, Oxford
12. Website GridGain: http://gridgain.com